Crate commonware_storage Copy item path

Settings

Help

Summary

Source

Persist and retrieve data from an abstract store.

Status

`commonware-storage` is ALPHA software and is not yet recommended for production use. Developers should expect breaking changes and occasional instability.

Modules

§

- archive
- A write-once key-value store optimized for low-latency reads.
- journal
- An append-only log for storing arbitrary data.
- metadata
- A key-value store optimized for atomically committing a small collection of metadata.

Module archive Copy item path

Settings

Help

Summary

Source

A write-once key-value store optimized for low-latency reads.

`Archive` is a key-value store designed for workloads where all data is written only once and is uniquely associated with both an `index` and a `key`.

Data is stored in `Journal` (an append-only log) and the location of written data is stored in-memory by both index and key (truncated representation using a caller-provided `Translator`) to enable single-read lookups for both query patterns over all archived data.

Notably, `Archive` does not make use of compaction nor on-disk indexes (and thus has no read nor write amplification during normal operation).

Format

`Archive` stores data in the following format:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+-
--+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 
|15 |16 |       ...       |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+-
--+---+---+---+---+---+
|          Index(u64)           |  Key(Fixed Size)   |
C(u32)     |     Data       |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+-
--+---+---+---+---+---+

C = CRC32(Key)
```

To ensure keys fetched using `Journal::get_prefix` are correctly read, the index and key are checksummed within a `Journal` entry (although the entire entry is also checksummed by `Journal`).

Uniqueness

`Archive` assumes all stored indexes and keys are unique. If the same key is associated with multiple `indices`, there is no guarantee which value will be returned. If the a key is written to an existing `index`, `Archive` will return an error.

Conflicts

Because a truncated representation of a key is only ever stored in memory, it is possible (and expected) that two keys will eventually be represented by the same truncated key. To handle this case, `Archive` must check the persisted form of all conflicting keys to ensure data from the correct key is returned. To support efficient checks, `Archive` keeps a linked list of all keys with the same truncated prefix:

```
struct Record {
    index: u64,

    next: Option<Box<Record>>,
  }
```

To avoid random memory reads in the common case, the in-memory index directly stores the first item in the linked list instead of a pointer to the first item.

`index` is the key to the map used to serve lookups by `index` that stores the location of data in a given `Blob` (selected by `section = index & section_mask` to minimize the number of open `Journals`):

```
struct Location {
    offset: u32,
    len: u32,
  }
```

If the `Translator` provided by the caller does not uniformly distribute keys across the key space or uses a truncated representation that means keys on average have many conflicts, performance will degrade.

Memory Overhead

`Archive` uses two maps to enable lookups by both index and key. The memory used to track each index item is `8 + 4 + 4` (where `8` is the index, `4` is the offset, and `4` is the length). The memory used to track each key item is `~truncated(key).len() + 16` bytes (where `16` is the size of the `Record` struct). This means that an `Archive` employing a `Translator` that uses the first `8` bytes of a key will use `~40` bytes to index each key.

Sync

`Archive` flushes writes in a given `section` (computed by `index & section_mask`) to `Storage` after `pending_writes`. If the caller requires durability on a particular write, they can call `sync`.

Pruning

`Archive` supports pruning up to a minimum `index` using the `prune` method. After `prune` is called on a `section`, all interaction with a `section` less than the pruned `section` will return an error.

Lazy Index Cleanup

Instead of performing a full iteration of the in-memory index, storing an additional in-memory index per `section`, or replaying a `section` of `Journal`, `Archive` lazily cleans up the in-memory index after pruning. When a new key is stored that overlaps (same truncated value) with a pruned key, the pruned key is removed from the in-memory index.

Single Operation Reads

To enable single operation reads (i.e. reading all of an item in a single call to `Blob`), `Archive` caches the length of each item in its in-memory index. While it increases the footprint per key stored, the benefit of only ever performing a single operation to read a key (when there are no conflicts) is worth the tradeoff.

Compression

`Archive` supports compressing data before storing it on disk. This can be enabled by setting the `compression` field in the `Config` struct to a valid `zstd` compression level. This setting can be changed between initializations of `Archive`, however, it must remain populated if any data was written with compression enabled.

Querying for Gaps

`Archive` tracks gaps in the index space to enable the caller to efficiently fetch unknown keys using `next_gap`. This is a very common pattern when syncing blocks in a blockchain.

Example

```
use commonware_runtime::{Spawner, Runner,
deterministic::Executor};
use commonware_storage::{journal::{Journal, Config as
JournalConfig}, archive::{Archive, Config,
translator::FourCap}};
use prometheus_client::registry::Registry;
use std::sync::{Arc, Mutex};

let (executor, context, _) = Executor::default();
```

```
executor.start(async move {
    // Create a journal
    let cfg = JournalConfig {
        registry: Arc::new(Mutex::new(Registry::default())),
        partition: "partition".to_string()
    };
    let journal = Journal::init(context, cfg).await.unwrap();

    // Create an archive
    let cfg = Config {
        registry: Arc::new(Mutex::new(Registry::default())),
        key_len: 8,
        translator: FourCap,
        section_mask: 0xffff_ffff_ffff_0000u64,
        pending_writes: 10,
        replay_concurrency: 4,
        compression: Some(3),
    };
    let mut archive = Archive::init(journal,
cfg).await.unwrap();

    // Put a key
    archive.put(1, b"test-key", "data".into()).await.unwrap();

    // Close the archive (also closes the journal)
    archive.close().await.unwrap();
   });
```

Modules

- translator

Structs

- Archive
- Implementation of `Archive` storage.
- Config
- Configuration for `Archive` storage.

Enums

- Error
- Errors that can occur when interacting with the archive.
- Identifier
- Subject of a `get` or `has` operation.

Traits

- Translator
- Translate keys into an internal representation used in `Archive`’s in-memory index.

Module journal

Copy item path

Settings

Help

Summary

Source

An append-only log for storing arbitrary data.

`Journal` is an append-only log for storing arbitrary data on disk with the support for serving checksummed data by an arbitrary offset. It can be used on its own to persist streams of data for later replay (serving as a backing store for some in-memory data structure) or as a building block for a more complex construction that prescribes some meaning to offsets in the log.

Format

Data stored in `Journal` is persisted in one of many `Blobs` within a caller-provided `partition`. The particular `Blob` in which data is stored is identified by a `section` number (`u64`). Within a `section`, data is appended to the end of each `Blob` in chunks of the following format:

```
+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 |    ...    | 8 | 9 |10 |11 |
+---+---+---+---+---+---+---+---+---+---+---+
|   Size (u32)  |   Data    |    C(u32)     |
+---+---+---+---+---+---+---+---+---+---+---+

C = CRC32(Data)
```

To ensure data returned by `Journal` is correct, a checksum (CRC32) is stored at the end of each item. If the checksum of the read data does not match the stored checksum, an error is returned. This checksum is only verified when data is accessed and not at startup (which would require reading all data in `Journal`).

Open Blobs

`Journal` uses 1 `Blob` per `section` to store data. All `Blobs` in a given `partition` are kept open during the lifetime of `Journal`. If the caller wishes to bound the number of open `Blobs`, they should group data into fewer `sections` and/or prune unused `sections`.

Offset Alignment

In practice, `Journal` users won't store `u64::MAX` bytes of data in a given `section` (the max `Offset` provided by `Blob`). To reduce the memory usage for tracking offsets within `Journal`, offsets are thus `u32` (4 bytes) and aligned to 16 bytes. This means that the maximum size of any `section` is `u32::MAX * 17 = ~70GB` bytes (the last offset item can store up to `u32::MAX` bytes). If more data is written to a `section` past this max, an `OffsetOverflow` error is returned.

Sync

Data written to `Journal` may not be immediately persisted to `Storage`. It is up to the caller to determine when to force pending data to be written to `Storage` using the `sync` method. When calling `close`, all pending data is automatically synced and any open blobs are closed.

Pruning

All data appended to `Journal` must be assigned to some `section` (`u64`). This assignment allows the caller to prune data from `Journal` by specifying a minimum `section` number. This could be used, for example, by some blockchain application to prune old blocks.

Replay

During application initialization, it is very common to replay data from `Journal` to recover some in-memory state. `Journal` is heavily optimized for this pattern and provides a `replay` method that iterates over multiple `sections` concurrently in a single stream.

Skip Reads

Some applications may only want to read the first `n` bytes of each item during `replay`. This can be done by providing a `prefix` parameter to the `replay` method. If `prefix` is provided, `Journal` will only return the first `prefix` bytes of each item and "skip ahead" to the next item (computing the offset using the read `size` value).

Reading only the `prefix` bytes of an item makes it impossible to compute the checksum of an item. It is up to the caller to ensure these reads are safe.

Exact Reads

To allow for items to be fetched in a single disk operation, `Journal` allows callers to specify an `exact` parameter to the `get` method. This `exact` parameter must be cached by the caller (provided during `replay`) and usage of an incorrect `exact` value will result in undefined behavior.

Example

```
use commonware_runtime::{Spawner, Runner,
deterministic::Executor};
use commonware_storage::journal::{Journal, Config};
use prometheus_client::registry::Registry;
use std::sync::{Arc, Mutex};

let (executor, context, _) = Executor::default();
executor.start(async move {
    // Create a journal
    let mut journal = Journal::init(context, Config{
        registry: Arc::new(Mutex::new(Registry::default())),
        partition: "partition".to_string()
    }).await.unwrap();

    // Append data to the journal
    journal.append(1, "data".into()).await.unwrap();

    // Close the journal
    journal.close().await.unwrap();
  });
```

Structs

- Config
- Configuration for `Journal` storage.
- Journal
- Implementation of `Journal` storage.

Enums

- Error
- Errors that can occur when interacting with `Journal`.

commonware_storage

Module metadataCopy item path

Settings

Help

Summary

Source

A key-value store optimized for atomically committing a small collection of metadata.

`Metadata` is a key-value store optimized for tracking a small collection of metadata that allows multiple updates to be committed in a single batch. It is commonly used with a variety of other underlying storage systems to persist application state across restarts.

Format

Data stored in `Metadata` is serialized as a sequence of key-value pairs in either a “left” or “right” blob:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+-
--+---+---+---+---+---+---+---+
| 0 | 1 |    ...    |15 |16 |17 |18 |19 |20 |21 |22 |23 |24 |
...   |50 |...|90 |91 |92 |93 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+-
--+---+---+---+---+---+---+---+
|   Timestamp (u128)    |  Key1 (u32)   | Len(V1) (u32) |
Value1     |...|  CRC32(u32)   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+-
--+---+---+---+---+---+---+---+

Len(V1) = Length of Value1
  ... = Other key-value pairs (Key2|VLen2|Value2,
  Key3|VLen3|Value3, ...)
```

To ensure the integrity of the data, a CRC32 checksum is appended to the end of the blob. This ensures that partial writes are detected before any data is relied on.

In the unlikely event that the current timestamp since the last `sync` is unchanged (as measured in nanoseconds), the timestamp is incremented by one to ensure that the latest update is always considered the most recent on restart.

Atomic Updates

To provide support for atomic updates, `Metadata` maintains two blobs: a “left” and a “right” blob. When a new update is committed, it is written to the “older” of the two blobs (indicated by the timestamp persisted). Writes to `Storage` are not atomic and may only complete partially, so we only overwrite the “newer” blob once the “older” blob has been synced (otherwise, we would not be guaranteed to recover the latest complete state from disk on restart as half of a blob could be old data and half new data).

Example

```
use commonware_runtime::{Spawner, Runner,
deterministic::Executor};
use commonware_storage::metadata::{Metadata, Config};
use prometheus_client::registry::Registry;
use std::sync::{Arc, Mutex};

let (executor, context, _) = Executor::default();
executor.start(async move {
    // Create a store
    let mut metadata = Metadata::init(context, Config{
        registry: Arc::new(Mutex::new(Registry::default())),
        partition: "partition".to_string()
    }).await.unwrap();
    // Store metadata
    metadata.put(1, "hello".into());
    metadata.put(2, "world".into());

    // Sync the metadata store (batch write changes)
    metadata.sync().await.unwrap();

    // Retrieve some metadata
    let value = metadata.get(1);

    // Close the store
    metadata.close().await.unwrap();
  });
```

Structs

- Config
- Configuration for `Metadata` storage.
- Metadata
- Implementation of `Metadata` storage.

Enums

- Error
- Errors that can occur when interacting with `Metadata`.

commonware_storage::journal

Struct Config

Copy item path

Settings

Help

Summary

Source

```
pub struct Config {
    pub registry: Arc<Mutex<Registry>>,
    pub partition: String,
}
```

Configuration for `Journal` storage.

Fields

`registry: Arc<Mutex<Registry>>`

Registry for metrics.

`partition: String`

The `commonware-runtime::Storage` partition to use for storing journal blobs.

Trait Implementations

Source

`impl Clone for Config`

Source

`fn clone(&self) -> Config`

Returns a copy of the value. Read more

1.0.0 · Source

`fn clone_from(&mut self, source: &Self)`

Performs copy-assignment from `source`. Read more

Auto Trait Implementations

`impl Freeze for Config`

`impl RefUnwindSafe for Config`

`impl Send for Config`

`impl Sync for Config`

`impl Unpin for Config`

`impl UnwindSafe for Config`

Blanket Implementations

Source

```
impl<T> Any for T

where
    T: 'static + ?Sized,
```

Source

```
impl<T> Borrow<T> for T

where
    T: ?Sized,
```

Source

```
impl<T> BorrowMut<T> for T

where
    T: ?Sized,
```

Source

```
impl<T> CloneToUninit for T

where
    T: Clone,
```

Source

```
impl<T> From<T> for T
```

Source

```
impl<T> Instrument for T
```

Source

§

```
impl<T, U> Into<U> for T

where
```

```
    U: From<T>,
```

Source

```
impl<T> Same for T
```

Source

```
impl<T> ToOwned for T

where
    T: Clone,
```

Source

```
impl<T, U> TryFrom<U> for T

where
    U: Into<T>,
```

Source

```
impl<T, U> TryInto<U> for T

where
    U: TryFrom<T>,
```

Source

```
impl<V, T> VZip<V> for T

where
    V: MultiLane<T>,
```

Source

```
impl<T> WithSubscriber for T
```

commonware_storage::journal

Struct Journal

Copy item path

Settings

Help

Summary

Source

```
pub struct Journal<B: Blob, E: Storage<B>> { /* private fields */
}
```

Implementation of `Journal` storage.

Implementations

Source

```
impl<B: Blob, E: Storage<B>> Journal<B, E>
```

Source

```
pub async fn init(runtime: E, cfg: Config) -> Result<Self, Error>
```

Initialize a new `Journal` instance.

All backing blobs are opened but not read during initialization. The `replay` method can be used to iterate over all items in the `Journal`.

Source

```
pub async fn replay(
    &mut self,
    concurrency: usize,
    prefix: Option<u32>,
) -> Result<impl Stream<Item = Result<(u64, u32, u32, Bytes),
Error>> + '_, Error>
```

Returns an unordered stream of all items in the journal.

Repair

If any corrupted data is found, the stream will return an error.

If any trailing data is found (i.e. misaligned entries), the journal will be truncated to the last valid item. For this reason, it is recommended to call `replay` before calling `append` (as data added to trailing bytes will fail checksum after restart).

Concurrency

The `concurrency` parameter controls how many blobs are replayed concurrently. This can dramatically speed up the replay process if the underlying storage supports concurrent reads across different blobs.

Prefix

If `prefix` is provided, the stream will only read up to `prefix` bytes of each item. Consequently, this means we will not compute a checksum of the entire data and it is up to the caller to deal with the consequences of this.

Reading `prefix` bytes and skipping ahead to a future location in a blob is the theoretically optimal way to read only what is required from storage, however, different storage implementations may take the opportunity to readahead past what is required (needlessly). If the underlying storage can be tuned for random access prior to invoking replay, it may lead to less IO.

Source

```
pub async fn append(&mut self, section: u64, item: Bytes) ->
Result<u32, Error>
```

Appends an item to `Journal` in a given `section`.

Warning

If there exist trailing bytes in the `Blob` of a particular `section` and `replay` is not called before this, it is likely that subsequent data added to the `Blob` will be considered corrupted (as the trailing bytes will fail the checksum verification). It is recommended to call `replay` before calling `append` to prevent this.

Source

```
pub async fn get_prefix(
    &self,
    section: u64,
    offset: u32,
    prefix: u32,
) -> Result<Option<Bytes>, Error>
```

Retrieves the first `prefix` bytes of an item from `Journal` at a given `section` and `offset`.

This method bypasses the checksum verification and the caller is responsible for ensuring the integrity of any data read.

Source

```
pub async fn get(
    &self,
    section: u64,
    offset: u32,
    exact: Option<u32>,
) -> Result<Option<Bytes>, Error>
```

Retrieves an item from `Journal` at a given `section` and `offset`.

If `exact` is provided, it is assumed the item is of size `exact` (which allows the item to be read in a single read). If `exact` is provided, the checksum of the data is still verified.

Source

```
pub async fn sync(&self, section: u64) -> Result<(), Error>
```

Ensures that all data in a given `section` is synced to the underlying store.

If the `section` does not exist, no error will be returned.

Source

```
pub async fn prune(&mut self, min: u64) -> Result<(), Error>
```

Prunes all `sections` less than `min`.

Source

```
pub async fn close(self) -> Result<(), Error>
```

Closes all open sections.

Auto Trait Implementations

```
impl<B, E> Freeze for Journal<B, E>
where
    E: Freeze,
```

```
impl<B, E> RefUnwindSafe for Journal<B, E>
where
    E: RefUnwindSafe,
    B: RefUnwindSafe,
```

```
impl<B, E> Send for Journal<B, E>
```

```
impl<B, E> Sync for Journal<B, E>
```

```
impl<B, E> Unpin for Journal<B, E>
where
    E: Unpin,
```

```
impl<B, E> UnwindSafe for Journal<B, E>
where
    E: UnwindSafe,
    B: RefUnwindSafe,
```

Blanket Implementations

Source

```
impl<T> Any for T
where
    T: 'static + ?Sized,
```

Source

```
impl<T> Borrow<T> for T
where
    T: ?Sized,
```

Source

```
impl<T> BorrowMut<T> for T
where
    T: ?Sized,
```

Source

```
impl<T> From<T> for T
```

Source

§

```
impl<T> Instrument for T
```

Source

```
impl<T, U> Into<U> for T
where
    U: From<T>,
```

Source

```
impl<T> Same for T
```

Source

```
impl<T, U> TryFrom<U> for T
```

```
where
    U: Into<T>,
```

Source

```
impl<T, U> TryInto<U> for T

where
    U: TryFrom<T>,
```

Source

```
impl<V, T> VZip<V> for T

where
    V: MultiLane<T>,
```

Source

```
impl<T> WithSubscriber for T
```